UNIT 2
Debugging

Bug:

A defect or a fault in a machine, plan or the like

- Debugger or debugging tool is a computer program used to test and debug other programs.

- Eg: Program trying to use instruction not available in current version of CPU, Attempting to access unavailable or protected memory results in trap (bug) in a program.

- Whenever trap (bug) occurs the program cannot normally continue and hence must be debugged.

- Bug is a term that didnt originate with programmers, but it is the most common terms in computing.

Introduction

- Because of the complexity involved in a software (in terms of interaction of components), techniques are devised to reduce connections between components, so that there are fewer pieces to interact with.
- Eg: Information hiding, abstraction and interfaces, language features are all devised for the same.

- Also there are techniques to ensure integrity of software design .
- Eg: program proofs, modelling, requirements analysis, formal verification

- In conclusion, all these have not changed the way software is built, successful only on small problems.
- Reality: There will always be errors that we find by testing and eliminate by debugging.

- Good programmers – spend as much time in debugging as writing so as to learn from mistakes.
- Every bug – teach us to prevent from similar bug happening again or to recognize it if it does.

- Debugging is hard, takes long and unpredictable amounts of time, so goal is avoid having to do much of it.

- How??
- Techniques that help reduce debugging time include good design, good style, boundary condition tests, assertions, sanity check in the code, defensive programming, well-designed interfaces, limited global data, checking tools.

- Role of language:
- To prevent bugs through language features.
- Some features like range checking on subscripts, restricted pointers or no pointers at all, garbage collection, string data types, typed I/O, strong type-checking.

- Some features that are prone to error are:
- goto statements, global variables, unrestricted pointers, automatic type conversions.

- Higher level languages makes simple bugs disappear automatically, makes it easier to create higher-level bugs.

Debuggers

- Compilers come with sophisticated debuggers, packaged as part of development environment that integrates creation and editing of source code, compilation, execution and debugging all into a single system.

- Debugging include graphical interfaces for stepping through a program one statement or function at a time, stop at particular lines or when a specific condition occurs.

- Debugger is invoked directly when a problem is known to exist. Some take over automatically when something unexpectedly goes wrong during program execution.

- Information needed to identify a bug :
  - Where the program was executing when it died
  - Examine the sequence of functions that were active (stack trace).
  - Display values of local and global variables.
  - Or else there can be breakpoints to re-run a failing program one step at a time and find first place where something went wrong.

  - Debuggers are system-dependent, so access to familiar debugger may not be there.

  - Multi-process, multi-threaded programs Operating systems, distributed systems must be often debugged by lower-level approaches.
  - Blind probing with a debugger is not likely to be productive.
  - Use the debugger to discover the state of the program when it fails, then think about how the failure could have happened.
  - Debugger can be of enormous value, certainly include it in your debugging toolkit.

Good Clues, Easy bugs

- Program crash, printing nonsense or seems to be running forever.
- Beginners and the experienced also tend to blame the compiler, the library or anything rather than their own code.
- Examine the evidence in the erroneous output and try to infer how it could have been produced.
- Look at any debugging output before the crash, try to get a stack trace (report that provides information about program subroutines) from a debugger.
- Debugging involves backwards reasoning. Think backward from result to discover the reasons.

- Look for familiar patterns:
- Bugs that you must have seen before has familiar pattern.
- Common bugs have distinctive signatures.

- ? int n;
- ? scanf("%d”, n);
- instead of
- int n;
- scanf ("%d”, &n) ;

- this typically causes attempt to access out-of-bounds memory when a line of input is read. Recognize the symptom instantly.

- Mismatched types and conversions in printf and scanf are an endless source of easy bugs.

```
?     int n = 1;
?     double d = PI;
?     printf("%d %f\n", d, n);
```

- Another common error is using %f instead of %lf to read a double with scanf.

- Some compilers catch such mistakes by verifying that the types of scanf and printf arguments match their format strings; if all warnings are enabled, for the printf above, the GNU compiler gcc reports that
- x.c:9: warning: i n t format, double arg (arg 2)
- x.c:9: warning: double format, different type arg (arg 3)
- Failing to initialize a local variable
- Result is garbage leftover from previous value stored in same memory location.
- Memory returned by allocators like malloc, realloc and new is likely to be garbage, be sure to initialize it.

- 2. Examine the most recent change:
- What was the last change?
- If you're changing only one thing at a time as a program evolves, the bug most likely is either in the new code or has been exposed by it.

- Looking carefully at recent changes helps to localize the problem. If bug appears in new version, then new code is part of the problem.

- Therefore preserve atleast the previous version of the program, so that you can compare behaviours.

- Keep record of changes and bugs fixed, so no need to rediscover information while trying to fix a bug.
- 3. Dont make the same mistake twice:
- See if you have made the same mistake elsewhere once you fix a bug.

- 4. Debug it now, not later:
- Don't ignore a crash when it happens; track it down right away, since it may not happen again until it's too late.

- A famous example occurred on the Mars Pathfinder mission. After the flawless landing in July 1997 the spacecraft's computers tended to reset once a day or so, and the engineers were baffled. Once they tracked down the problem, they realized that they had seen that problem before.
- During pre-launch tests the resets had occurred, but had been ignored because the engineers were working on unrelated problems.
- So they were forced to deal with the problem later when the machine was tens of millions of miles away and much harder to fix.
- 5. Get a stack trace:
- Although debuggers can probe running programs, one of their most common uses is to examine the state of a program after death.
- The source line number of the failure, often part of a stack trace, is the most useful single piece of debugging information;

- improbable values of arguments are also a big clue (zero pointers, integers that are huge when they should be small, or negative when they should be positive, character strings that aren't alphabetic).
- A debugger can also be used to display values of local or global variables that will give additional information about what went wrong.
- 0 strcmp(Oxla2, Oxlc2) ["strcmp.s":31]
- 1 scmp(p1 = 0x10001048, p2 = 0x1000105c) ["badqs.c":131
- 2 qst(0x10001048, 0x10001074, Ox400b20, 0x4) ["qsort.cc:147]
- 3 qsort(0x10001048, Oxlc2, 0x4, Ox400b20) ["qsort.c":631
- 4 mai n() [" badqs . c" : 451
- 5 --sit a r t () [" c r t l t i nit .s " : 131

- 6. Read before typing:
- One effective but under-appreciated debugging technique is to read the code very carefully and think about it for a while without making changes.

- But chances are that you don't know what's really broken and will change the wrong thing, perhaps breaking something else.

- A listing of the critical part of program on paper can give a different perspective than what you see on the screen, and encourages you to take more time for reflection.

- Take a break for a while; sometimes what you see in the source code is what you meant rather than what you wrote, and an interval away from it can soften your misconceptions and help the code speak for itself when you return.

- Resist the urge to start typing; thinking is a worthwhile alternative.

- 7. Explain your code to someone else.
- Another effective technique is to explain your code to someone else. This will often cause you to explain the bug to yourself.

- Sometimes it takes no more than a few sentences, followed by an embarrassed "Never mind, I see what's wrong. Sorry to bother you." This works remarkably well; you can even use non-programmers as listeners.

- One university computer center kept a teddy bear near the help desk. Students with mysterious bugs were required to explain them to the bear before they could speak to a human counselor.

No Clues, Hard Bugs

- "I haven't got a clue. What on earth is going on?" If you really haven't any idea what could be wrong, life gets tougher.

- 1. Make the bug reproducible.
- The first step is to make sure you can make the bug appear on demand. It's frustrating to chase down a bug that doesn't happen every time.
- Spend some time constructing input and parameter settings that reliably cause the problem, then wrap up the recipe so it can be run with a button push or a few keystrokes.
- If it's a hard bug, you'll be making it happen over and over as you track down the problem, so you'll save yourself time by making it easy to reproduce.
- If the bug can't be made to happen every time, try to understand why not. Does some set of conditions make it happen more often than others?

- 2. Divide and Conquer
- Can the input that causes the program to fail be made smaller or more focused?
- Narrow down the possibilities by creating the smallest input where the bug still shows up.

- What changes make the error go away?
- Try to find crucial test cases that focus on the error.
- Each test case should aim at a definitive outcome that confirms or denies a specific hypothesis about what is wrong.
- The same binary search process can be used on the program text itself:
- Eliminate some part of the program that should have no relationship to the bug and see if the bug is still there.

- 3. Study the numerology of failures.
- Studying the patterns of numbers related to the failure pointed us right at the bug.
- Elapsed time?

- A couple of minutes of mystification, five minutes of looking at the data to discover the pattern of missing characters, a minute to search for likely places to fix, and another minute to identify and eliminate the bug.

- 3. Display output to localize your search.
- If you don't understand what the program is doing, adding statements to display more information can be the easiest, most cost effective way to find out.
- Put them in to verify your understanding or refine your ideas of what's wrong.
- Display messages in a compact fixed format so they are easy to scan by eye or with programs like the pattern-matching tool grep.

- 4. Write self-checking code.
- If more information is needed, you can write your own check function to test a condition, dump relevant variables. and abort the program.
- 5. Write a logfile.
- Another tactic is to write a logJle containing a fixed-format stream of debugging output.
- When a crash occurs. the log records what happened just before the crash.

```
[Sun Dec 27 16:19:24 19981
HTTPd: access to /usr/local /httpd/cgi -bi n/test. html
failed for ml.cs.bel1-labs.com,
reason : client denied by server (CGI non-executable)
from http://m2.cs.bell-labs.com/cgi-bin/test.pl
```

- 6. Draw a picture.
- Sometimes pictures are more effective than text for testing and debugging.
- Scatter plots display misplaced values more effectively than columns of numbers.
- If you don't understand what's happening inside your program, try annotating the data structures with statistics and plotting the result.
- 7. Use tools.
- Make good use of the facilities of the environment where you are debugging.
- 8. Keep records.
- The act of writing will help you remember the problem the next time something similar comes up, and will also serve when you're explaining it to someone else.

Last Resorts

- Final course of action.
- What do you do if none of this advice helps?
- This may be the time to use a good debugger to step through the program.

- Example, programmers often forget that & and | have lower precedence than == and ! =. They write

```
?     if (x & 1 == 0)
?         ...
```

- and can't figure out why this is always false.

```
?     while ((c == getchar()) != EOF)
?         if (c = '\n')
?             break;
```

Or extra code is left behind during editing:

```
?     for (i = 0; i < n; i++);
?         a[i++] = 0;
```

Or hasty typing creates a problem:

```
?     switch (c) {
?         case '<':
?             mode = LESS;
?             break;
?         case '>':
?             mode = GREATER;
?             break;
?         defualt:
?             mode = EQUAL;
?             break;
?     }
```

Sometimes the error involves arguments in the wrong order in a situation where type-checking can't help, like writing

```
?     memset(p, n, 0);     /* store n 0's in p */
```

instead of

```
      memset(p, 0, n);     /* store n 0's in p */
```

- If you can't find a bug after considerable work, take a break.
- Clear your mind, do something else. Talk to a friend and ask for help. The answer might appear out of the blue, but if not, you won't be stuck in the same rut in the next debugging session.

- Occasionally hardware itself goes bad.
- The tloating-point flaw in the 1994 Pentium processor that caused certain computations to produce wrong answers was a highly publicized and costly bug in the design of the hardware.

- The problem was eventually traced to a failure of the floating-point unit in one of the processors.

Non-reproducible Bugs

- Bugs that won't stand still are the most difficult to deal with, and usually the problem isn't as obvious as failing hardware.
- Check whether all variables have been initialized; you may be picking up a random value from whatever was previously stored in the same memory location.
- If the bug changes behavior or even disappears when debugging code is added, It may be a memory allocation error-somewhere you have written outside of allocated memory,

- If the crash site seems far away from anything that could be wrong, the most likely problem is overwriting memory by storing into a memory location that isn't used until much later.

- Sometimes this is a dangling pointer problem, where a pointer to a local variable is inadvertently returned from a function, then used.

- When a program works for one person but fails for another, something must depend on the external environment of the program.

- This might include files read by the program, file permissions, environment variables, search path for commands, defaults, or startup files.

Debugging Tools

- Many programs that are part of standard toolkit, are written to help find a particular bug or to analyse a specific program other than the debuggers.
- These variety of programs can help us wade voluminous output to select important bits, find anomalies, or rearrange data.

Other People's Bugs

- Realistically, most programmers do not have the fun of developing a brand new system from the ground up.
- Instead, they spend much of their time using, maintaining, modifying and thus, inevitably, debugging code written by other people.
- When debugging others' code, everything that we have said about how to debug your own code applies.
- Before starting, though, you must first acquire some understanding of how the program is organized and how the original programmers thought and wrote.

- Text-search programs like grep can find all the occurrences of names.
- Cross-referencers give some idea of the program's structure.
- A display of the graph of function calls is valuable if it isn't too big.
- Stepping through a program a function call at a time with a debugger can reveal the sequence of events.

- A revision history of the program may give some clues by showing what has been done to the program over time.
- Frequent changes are often a sign of code that is poorly understood or subject to changing requirements. and thus potentially buggy.
- Sometimes you need to track down errors in software you are not responsible for and do not have the source code for.
- If you think that you have found a bug in someone else's program, the first step is to make absolutely sure it is a genuine bug, so you don't waste the author's time and lose your own credibility.
- When you find a compiler bug, make sure that the error is really in the compiler and not in your own code.

- For example, whether a right shift operation fills with zero bits (logical shift) or propagates the sign bit (arithmetic shift) is unspecified in C and C++, so novices sometimes think it's an error if a construct like

```
? i = -1;
? printf ("%d\n”, i >> 1) ;
```

- yields an unexpected answer.
- But this is a portability issue, because this statement can legitimately behave differently on different systems.
- Try your test on multiple systems and be sure you understand what happens; check the language definition to be sure.

- Make sure the bug is new.
- Do you have the latest version of the program?
- Is there a list of bug fixes? Most software goes through multiple releases; if you find a bug in version 4.0b1, it might well be fixed or replaced by a new one in version 4.04b2.
- Finally, put yourself in the shoes of the person who receives your report. You want to provide the owner with as good a test case as you can manage.
- It's not very helpful if the bug can be demonstrated only with large inputs, or an elaborate environment, or multiple supporting files.
- Strip the test down to a minimal and self contained case.
- Include other information that could possibly be relevant, like the version of the program itself. and of the compiler. operating system. and hardware.